Fun Facts about the 8088

Compiled by Chris Peters

1. Comparing a register

The fastest and smallest way to compare a 16 bit register to zero is to OR it with itself, e.g.

```
OR      BX,BX           ; 2 bytes, 3 clocks
JGE     BXisPositive
```

this is much better than comparing it with zero, e.g.

```
CMP     BX,0    ; 3 bytes, 4 clocks (bush league)
```

For the ultimate in comparing with zero, try to use the CX register. The 8088 contains the single instruction:

```
JCXZ    CXisZero        ; jump if CX is zero
```

This instruction makes a short jump if CX is zero.

To destructivly test for 1 or -1, use the DEC or INC instructions:

```
DEC     DX              ; 1 byte, 3 clocks
JZ      DXisOne         ; if zero, DX was 1
```

or

```
INC     DX              ; 1 byte, 3 clocks
JZ      DXisMinusOne    ; if zero, DX was -1
```

The LOOP instruction is just a fancy way of writing:

```
DEC     CX
JNZ     CXisNotZero
```

The difference is that LOOP is 1 byte smaller and 2 clocks faster. The LOOP instruction can be used to compare CX with multiple values as follows:

```
LOOP    CXisNotOne      ; If CX = 1 then...
...                     ; ...do this code, else...
```

```
CXisNotOne:
    LOOP    CXisNotTwo          ; if CX = 2 then...
    ...                         ; ...do this code, else...
CXisNotTwo:
    LOOP    CXisNotThree        ; if CX = 3 then...
    ...                         ; ...do this code, else etc.
```

Its possible to check if a signed number is in the range 0-n with a single comparison to n:

```
    CMP     DX,639              ; if <0 or >639...
    JA      OutOfRange          ; ...its out of range
```

This is smaller and faster than:

```
    OR      DX,DX               ; Never CMP DX,0!
    JL      OutOfRange          ; If negative, its out of range
    CMP     DX,639
    JG      OutOfRange          ; if greater, its out of range
```

You cannot compare a segment register. To do so copy it to a register or memory location, then compare it.

2. Setting a register to zero

To set a register to zero, the smallest, fastest way is to XOR it with itself, e.g.

```
    XOR     BX,BX               ; 2 bytes, 2 clocks
```

is smaller and faster than:

```
    MOV     DX,0                ; 3 bytes, 4 clocks
```

There is one side affect: MOVing does not affect the flags, but XORing does. The 8088 aficianado will only move a zero into a register in the rare cases where the flags must be preserved.

3. Incrementing, Decrementing

It is smaller to increment or decrement a 16 bit register then an 8 bit register, so if it doesnt matter, use a 16 bit register, e.g.:

```
INC     DX                  ; 1 byte, 3 clocks
```

is smaller than:

```
INC     DL                  ; 2 bytes, 3 clocks
```

Same thing goes for decrementing, its smaller to use a 16 bit register.

Its smaller (but not faster) to increment a register twice then to add 2 to it, e.g.

```
INC     DX                  ; 1 byte,  3 clocks (total)
INC     DX                  ; 2 bytes, 6 clocks
```

is smaller and slower than:

```
ADD     DX,2                ; 3 bytes, 4 clocks
```

One side affect: INC and DEC do not affect the carry flag, ADD does.

4. If Then Else

When confronted with an If, Then, Else problem the assembly language programmer will often write it as Else, If, Then. For example, a sample problem might be to return 80H in the DX register if AL<5, otherwise return zero in DX. Using If, Then, Else produces:

```
    CMP     AL,5                ; Is AL less than 5?
    JB      ALisBelow5          ; yes...
    XOR     DX,DX               ; Set DX to zero if AL>=5
    JMP     SHORT Continue      ; proceed
ALisBelow5:
    MOV     DX,80H              ; Set DX to 80H if AL<5
Continue:
```

Using Else, If, Then:

```
    MOV     DX,80H          ; Set DX to 80H if AL<5
    CMP     AL,5            ; Is AL less than 5?
    JB      ALisBelow5      ; yes...
    XOR     DX,DX           ; Set DX to zero if AL>=5
ALisBelow5:
```

The idea is to do the work for the most likely case, then do the comparision. If you were right you're all done, if not then do the other case.

5. Copying strings

One of the simplest ways to copy a null terminated string is as follows:

```
;
;  DS:SI points at source
;  ES:DI points at destination
;
CopyString:
    LODSB                   ; read character into AL, inc SI
    STOSB                   ; store charcter, increment DI
    OR      AL,AL           ; was it the null
    JNZ     CopyString      ; no, repeat
```

A fast way to copy a string where the count is known is

```
    SHR     CX,1    ; divide count by two (2 bytes = 1 word)
    REP     MOVSW   ; move the first part fast
    JNC     Even    ; No carry if count was even
    MOVSB           ; move the odd byte
Even:
```

The REP instruction checks to see if the loop count in CX is zero before starting, there is no need to check it beforehand.

6. Testing long pointer for null

Sometimes its necessary to check if a long pointer is null before using it. A sequence of code that works well is:

```
    LES     DI,[LongPointer]        ; get the long pointer
    MOV     CX,ES                   ; copy ES to CX
    JCXZ    PointerIsNull           ; if zero, dont use it
```

This assumes segment zero is an invalid pointer.

7. Exchanging

To move a register to or from AX takes 2 bytes and 2 clocks

```
MOV     AX,DX           ; 2 bytes, 2 clocks
```

However, to exchange a register with the AX register takes 1 byte and 3 clocks:

```
XCHG    AX,DX           ; 1 byte, 3 clocks
```

A clear case where its possible to optimize for size or speed. This optimization is only available with the AX register. A full understanding is actually more complex. Although XCHG takes 3 clocks it
is often faster because it only uses one byte of the prefetch queue.

8. Testing bits

The 8088 contains an instruction that allows you to test various bits. It works by doing a non destructive AND operation.

```
TEST    AX,0800H        ; 4 bytes, 5 clocks
JZ      BitIsOff
```

If one of the bytes is zero (a common occurence) this can be optimized to:

```
TEST    AH,08H          ; 3 bytes, 4 clocks
JZ      BitIsOff
```

The best way to test the hi bit is:

```
OR      AX,AX           ; 2 bytes, 2 clocks
JNS     HiBitIsOff      ; jump not signed (hi bit zero)
```

A destructive way to test the low order bit is to shift it to the right into the carry flag:

```
    SHR     AX,1            ; 2 bytes, 2 clocks
    JNC     LoBitIsOff
```

You can test multiple bits with the TEST instruction:

```
    TEST    BL,11000000b    ; check 2 highest bits
    JZ      BothAreZero     ; if zero, both are zero
```

Sometimes you want to jump if either are zero instead of both zero, this can usually be accomplished by the NOT instruction and reversing the sense of the jump instruction:

```
    NOT     BL              ; reverse the bits
    TEST    BL,11000000b    ; check 2 highest bits
    JNZ     EitherAreZero   ; if either were zero, then this
                            ; is non zero
```

9. Absolute value

A fascinating way to get the absolute value of the AX register was discovered by Marlin Eller:

```
    CWD                 ; replicate hi order bit of AX into DX
    XOR     AX,DX       ; do a 1's complement or do nothing
    SUB     AX,DX       ; add 1 to get a 2's complement
```

The boring method does not affect the DX register and can be used on any register:

```
    OR      BX,BX       ; never CMP BX,0!
    JGE     NotNeg      ; if negative...
    NEG     BX          ; ...make it positive
NotNeg:
```

The boring method empties the prefetch queue with the JGE instruction, making it much slower.

10. Length of null terminated string

To get the length of a null terminated string, scan for the null at the end with a starting count of -1

```
;
;  ES:DI points at null terminated string
;
    XOR     AL,AL           ; look for null    2 bytes (total)
    MOV     CX,-1           ; CX = -1          5 bytes
    REPNE   SCASB           ; CX = -len-2      7 bytes
    NOT     CX              ; CX =  len+1      9 bytes
    DEC     CX              ; CX =  len       10 bytes
```

This count does not include the null at the end. If you want it to include the null, just delete the final DEC CX. The use of the NOT instruction is quite interesting here.

11. Returning flags

The 8088 contains instructions for setting (STC) and clearing (CLC) the carry flag. To set the zero flag, simply compare some register with itself:

```
    CMP     DX,DX               ; set zero flag
```

To clear the zero flag, OR the stack pointer with itself:

```
    OR      SP,SP               ; clear zero flag
```

This is making the safe assumption that the stack pointer is not zero.

12. Shifting

Variable count shifting is slow on the 8088. Its faster to shift twice then to set a count of 2:

```
    SHR     AX,1    ; 2 bytes, 2 clocks  (total)
    SHR     AX,1    ; 4 bytes, 4 clocks
```

is much faster than:

```
    MOV     CL,2    ; 2 bytes,  4 clocks  (total)
    SHR     AX,CL   ; 4 bytes, 20 clocks!
```

Variable count shifting is slower when shifting less than 5 bits, after that the prefetch queue makes variable shift counts faster.

13. Multiply and Divide

The multiply and divide instruction are some of the slowest instructions on the 8088. To give some perspective, a register to register MOV instruction takes 2 clocks, while a signed divide (IDIV) using registers can take 184 clocks.

If your goal is to write fast 8088 code, multiplying by constants can usually be done as a series of shifts and adds:

```
;
;  Multiply the AX register by 10
;
    SHL     AX,1            ; AX = AX *  2     ( 2 clocks)
    MOV     BX,AX           ; BX = AX *  2     ( 4 clocks)
    SHL     AX,1            ; AX = AX *  4     ( 6 clocks)
    SHL     AX,1            ; AX = AX *  8     ( 8 clocks)
    ADD     AX,BX           ; AX = AX * 10     (11 clocks)
```

A multiply would be more than 10 times slower, but would take fewer bytes. Multiply is useful when neither argument is constant or you need to save bytes.

Mark Zbikowski uses this method to divide the AX register by 512:

```
    XCHG    AL,AH           ; divide AX by 256
    SHR     AL,1            ; divide AX by 512
    CBW                     ; AL < 128, so this sets AH to 0
```

14. Converting bytes to segments

To convert a byte count to a paragraph count try:

```
;
;  DX contains a byte count
;
    ADD     DX,15           ; round up to next paragraph
    MOV     CL,4            ; 2^4 = 16 bytes per paragraph
    SHR     DX,CL           ; divide by 16 by shifting 4 times
```

DX now contains a paragraph count. This assumes the value in DX is less than 0FFF1H. To cover the extended case:

```
ADD     DX,15           ; round up to next paragraph
RCR     DX,1            ; divide by 2, including carry
MOV     CL,3            ; 2^3 = 8
SHR     DX,CL           ; divide by a total of 16
```

15. Call, Return, Jump

A near call followd by a near return can always be replaced with a near jump:

```
JMP  NearProc  ; 3 bytes, 15 clocks
```

is smaller and much faster than:

```
CALL NearProc  ; 3 bytes, 19 clocks (total)
RET            ; 4 bytes, 35 clocks
```

Its often possible to eliminate the JMP entirely by moving the subroutines adjacent to each other.

Conditional jumps on the 8088 are always short, i.e. the destination must be within -128 to 127 of the instruction pointer. It seems every time a single line of new code is added some conditional jump becomes out of range. One technique to get around this is to find a similiar conditional jump to jump to:

```
   JC      OutOfRange      ; I want to jump to disk error...
   ...                     ; ...but its too far away, so...

OutOfRange:
   JC      DiskError       ; I jump to this test for carry
```

Although this is not in the scope of this document, out of range jumps are usually the 8088 telling you that your subroutines have grown too large and should be broken up.

16. Multiple Entry Points

An old 8080 trick involving multiple entry points can be adapted to the 8088. Instead of doing this:

```
Entry1:
    MOV     AL,1                    ; 2 bytes (total)
    JMP     SHORT EntryCommon       ; 4 bytes
Entry2:
    MOV     AL,2                    ; 6 bytes
    JMP     SHORT EntryCommon       ; 8 bytes
Entry3:
    MOV     AL,3                    ;10 bytes
EntryCommon:
```

The hearty and brave will do this:

```
Entry1:
    MOV     AL,1                    ; 2 bytes (total)
    DB      03DH                    ; 3 bytes
Entry2:
    MOV     AL,2                    ; 5 bytes
    DB      03DH                    ; 6 bytes
Entry3:
    MOV     AL,3                    ; 8 bytes
EntryCommon:                        ; flags are modified
```

The DB 03DH is the opcode for a CMP AX,xx. In this case the bogus CMP AX's are used to swallow up the MOV AL,x that follow.

17. Assertion Macros

When using these advanced techniques its important not to expose yourself to bugs caused by changing constants in your program. For instance, in the Multiply and Divide section of this document there is a code to quickly multiply by ten. If the constant should later change from ten to twelve this code would no longer work. An assertion macro would flag this code as being in error, saving many hours of needless debugging. In cannot be stressed to strongly that advanced 8088 programming requires liberal use of assertion macros and extra documentation.